WEST Search History

Hide Items | Restore | Clear | Cancel

DATE: Monday, February 07, 2005

Hide?	Set Name	Query	Hit Count
	DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=ADJ		
☐	L13	20010330	30
☐	L12	L8 and l3	44
☐	L11	L8 and l3	44
☐	L10	L8 and l2	1
☐	L9	L8 and l4	1
☐	L8	(first adj buffer) same (second adj buffer)	7419
☐	L7	L6 and l4	1
☐	L6	request near8 buffer near8 retrieve	233
☐	L5	L4 and((buffer or buffering) near8 requst)	0
☐	L4	L3 and l2	34
☐	L3	multi adj (thread or threading or threaded)	5963
☐	L2	inter-thread adj2 communications	73
	DB=USPT; PLUR=YES; OP=ADJ		
☐	L1	5630074.pn.	1

END OF SEARCH HISTORY

Previous Doc Next Doc Go to Doc#
First Hit Fwd Refs

☐ Generate Collection

L13: Entry 27 of 30 File: USPT Feb 16, 1999

DOCUMENT-IDENTIFIER: US 5872909 A
TITLE: Logic analyzer for software

Application Filing Date (1):
19950607

Brief Summary Text (17):
The present invention is also useful for embedded software. Regardless of the embedded software is real-time, the user embedded software has a need to monitor it. Embedded software is not often easily accessible, yet it is desirable to know how it is functioning for debugging, benchmarking and other purposes. Another application where a logic analyzer function would be beneficial is for multi-threaded or multi-tasking software. A snapshot at a given time, such as on failure, of the system is of limited usefulness where different tasks are swapped in and out. A history of what has occurred, which the present invention provides, is a very useful and powerful debugging and post-mortem analysis tool.

Detailed Description Text (73):
In a preferred embodiment, the event buffer is implemented as a double buffer. Such a scheme allows the uploading and emptying of the first buffer to the host to proceed asynchronously with event logging into the second buffer. There is, therefore, no delay in the target pending the availability of a (once full) buffer for logging. An event task is dedicated to emptying the buffer. Event logging must take into account that this task must be switched in and that switching itself generates data for the buffer.

Detailed Description Text (148):
Even though a preferred environment of the invention is real-time, the invention is not limited to real-time environments. Rather, the invention is useful wherever an analysis of a dynamic or multi-threaded software system and/or its applications is desirable.

Previous Doc Next Doc Go to Doc#

Previous Doc Next Doc Go to Doc#
First Hit Fwd Refs

Generate Collection

L13: Entry 21 of 30 File: USPT Oct 10, 2000

DOCUMENT-IDENTIFIER: US 6131126 A
TITLE: Method and apparatus for space-efficient inter-process communication

Application Filing Date (1):
19980127

Brief Summary Text (5):
Inter-Process communication is a fundamental part of modem day computer system design. Inter-process communication is typically facilitated via a calling scheme in a kernel of the computer system which manages communication between a client and server process. One of the problems associated with such inter-process communication is that typically, when a client task invokes a server task, the client task allocates a certain amount of memory to pass arguments (parameters) to the server task, and that memory is typically used until any arguments are returned from the server. That is, even though control and processing has been passed to the server application, memory is still consumed in the client application until return arguments are returned from the server. Thus, a buffer is allocated which is not used for a large portion of the time in which the server task has been passed control by the client. This especially is an issue in multi-threaded environments wherein a plurality of buffers are allocated, one for each thread. Multiple buffers remain allocated and, for the most part, stay unused for the duration of each thread, unnecessarily consuming memory resources.

Brief Summary Text (8):
While server process 120 is active, after the invocation by kernel 100. buffer 111 is still allocated in client 110. In certain prior art applications, buffers for marshaling arguments are on the order of five kilobytes in length. The client maintains this memory area open and accessible for the duration of server 120's execution. Upon completion of execution of server 120, a reverse of the client/server calling process described above is performed wherein the server uses its own buffer 121 for marshaling arguments into and a reference is passed to the area to kernel 100. Eventually, return arguments are within the original buffer area 111 contained within client 110. It is only at this time, in typical prior art systems, that the buffers 111 and 121 in both the client and the server are deallocated. Thus, the buffers are allocated and are idle for a long time in which kernel 100 and server process 120 are active and perhaps idle (e.g., awaiting I/O servicing). This is an unnecessary consumption of memory resources. Moreover, in multi-threaded environments, client 110 and server 120 may allocate numerous buffers such as 111 and 121, and maintain these in an allocated state while waiting for control to be returned by their corresponding called processes. This results in a very large and unnecessary consumption of memory resources.

Detailed Description Text (12):
Thus, buffers providing communication between the client and the server are dynamically allocated, on an "as needed" basis. Once the buffer is no longer required for communication between the two processes, it is deallocated via clearing of an "allocated buffer" flag. This is in contrast to the prior art which waits for a return from a second process (e.g., the server) until a buffer is deallocated. In multi-threaded environments, such as those in common use in modern

computer systems, the allocation of separate buffers for each call thread and the maintenance of these buffers during the call in an allocated state, consumes large amounts of memory resources. Typically, in such prior an circumstances, for the duration of the call to the server process these buffers are unused. Thus, the present invention provides a more efficient means for inter-process communication by avoiding the consumption of large amounts of memory by the allocation of buffers for multiple threads which are not required during execution of the second process (e.g., server process 320). Thus, implemented embodiments of the present invention use memory much more efficiently than such inter-process communication in the prior art.

CLAIMS:

1. A computer-implemented method in a computer system of inter-process communication comprising the following steps:

a. a first procedure allocating a first buffer in a first memory space shared by said first procedure and a second procedure;

b. said first procedure marshaling arguments for calling of a third procedure in said first buffer;

c. said first procedure calling said third procedure via said second procedure and passing a first reference to said first buffer in said first memory space to said second procedure;

d. said second procedure detecting said call of said third procedure by said first procedure, wherein said second procedure creating a thread and allocating a second buffer in a second memory space shared by said second procedure and said third procedure;

e. said second procedure referencing said first buffer and copying said marshaled arguments contained in said first buffer into said second buffer;

f. said second procedure calling said third procedure and passing a second reference to said second buffer to said third procedure, wherein said third procedure upon receiving said second reference, referencing and unmarshaling said marshaled arguments in said second buffer, said third procedure deallocating said second buffer while said second procedure maintains said thread.

9. An apparatus for inter-process communication comprising:

a. first circuitry for allocating a first buffer in a first memory space shared by a first procedure and a second procedure;

b. second circuitry for marshaling arguments in said first buffer referenced by said first procedure for communicating with a third procedure;

c. third circuitry for indicating a first message for said second procedure and passing a first reference to said first buffer in said first memory space from said first procedure to said second procedure;

d. fourth circuitry for detecting said indicating of said first message by said first procedure to said second procedure;

e. fifth circuitry for creating a thread upon detecting said indicating of said first message and allocating a second buffer in a second memory space shared by said second procedure and said third procedure;

f. sixth circuitry for referencing said first buffer and copying said marshaled

arguments contained in said first buffer into said second buffer;

g. seventh circuitry for indicating a second message for said third procedure and passing a second reference to said second buffer to said third procedure;

h. eighth circuitry for detecting said indicating of said second message by said second procedure to said third procedure;

i. ninth circuitry for referencing and unmarshaling said marshaled arguments in said second buffer;

j. tenth circuitry for deallocating said second buffer while said second procedure maintains said thread.

16. A computer system comprising:

a. a first allocation circuit for allocating a first buffer in a first memory space shared by a first procedure and a second procedure;

b. a first marshaling circuit for enabling said first procedure to marshal arguments for communicating with a third procedure;

c. a first message indication circuit for indicating a first message from said first procedure to said second procedure and passing a first reference to said first buffer in said first memory space to said second procedure;

d. a first message detection circuit for enabling said second procedure to detect said indicating of said first message by said first procedure;

e. a second allocation circuit for creating a thread upon detecting said indicating of said first message and allocating a second buffer in a second memory space shared by said second procedure and said third procedure;

f. a first referencing circuit for enabling said second procedure to reference said first buffer and copy said marshaled arguments contained in said first buffer into said second buffer;

g. a second message indication circuit for indicating a second message from said second procedure to said third procedure and passing a second reference to said second buffer to said third procedure;

h. a second message detection circuit for enabling said third procedure to detect said indicating of said second message by said second procedure;

i. a first unmarshaling circuit for enabling said third procedure to unmarshal said marshaled arguments in said second buffer;

j. a first deallocation circuit for deallocating said second buffer while said second procedure maintains said thread.

18. The computer system of claim 16 wherein said first referencing circuit further deallocates said first buffer upon completing said copying of said marshaled arguments contained in said first buffer into said second buffer.

Previous Doc Next Doc Go to Doc#

Previous Doc Next Doc Go to Doc#
First Hit Fwd Refs

Generate Collection

L13: Entry 26 of 30 File: USPT Mar 9, 1999

DOCUMENT-IDENTIFIER: US 5881286 A
TITLE: Method and apparatus for space efficient inter-process communications

Application Filing Date (1):
19970212

Brief Summary Text (5):
Inter-Process communication is a fundamental part of modern day computer system design. Inter-process communication is typically facilitated via a calling scheme in a kernel of the computer system which manages communication between a client and server process. One of the problems associated with such inter-process communication is that typically, when a client task invokes a server task, the client task allocates a certain amount of memory to pass arguments (parameters) to the server task, and that memory is typically used until any arguments are returned from the server. That is, even though control and processing has been passed to the server application, memory is still consumed in the client application until return arguments are returned from the server. Thus, a buffer is allocated which is not used for a large portion of the time in which the server task has been passed control by the client. This especially is an issue in multi-threaded environments wherein a plurality of buffers are allocated, one for each thread. Multiple buffers remain allocated and, for the most part, stay unused for the duration of each thread, unnecessarily consuming memory resources.

Brief Summary Text (8):
While server process 120 is active, after the invocation by kernel 100, buffer 111 is still allocated in client 110. In certain prior art applications, buffers for marshaling arguments are on the order of five kilobytes in length. The client maintains this memory area open and accessible for the duration of server 120's execution. Upon completion of execution of server 120, a reverse of the client/server calling process described above is performed wherein the server uses its own buffer 121 for marshaling arguments into and a reference is passed to the area to kernel 100. Eventually, return arguments are within the original buffer area 111 contained within client 110. It is only at this time, in typical prior art systems, that the buffers 111 and 121 in both the client and the server are deallocated. Thus, the buffers are allocated and are idle for a long time in which kernel 100 and server process 120 are active and perhaps idle (e.g., awaiting I/O servicing). This is an unnecessary consumption of memory resources. Moreover, in multi-threaded environments, client 110 and server 120 may allocate numerous buffers such as 111 and 121, and maintain these in an allocated state while waiting for control to be returned by their corresponding called processes. This results in a very large and unnecessary consumption of memory resources.

CLAIMS:

1. A computer-implemented method in a computer system of inter-process communication comprising the following steps:

a. a client procedure allocating a first buffer in a first memory space shared by said client procedure and a kernel procedure;

b. said client procedure marshaling arguments for said calling of a remote procedure in said first buffer;

c. said client procedure calling said remote procedure via said kernel procedure and passing a first reference to said first buffer in said first memory space to said kernel procedure;

d. said kernel procedure detecting said call of said remote procedure by said client procedure and allocating a second buffer in a second memory space shared by said kernel procedure and said server procedure;

e. said kernel procedure referencing said first buffer and copying said arguments contained in said first buffer into said second buffer;

f. said kernel procedure deallocating said first buffer;

g. said kernel procedure calling said remote procedure and passing a second reference to said second buffer to said remote procedure;

h. said remote procedure detecting said calling of said remote procedure and unmarshaling said arguments in said second buffer;

i. said remote procedure deallocating said second buffer and executing

wherein said kernel procedure initiates said deallocation of said first buffer:

(1) upon completion of said copying of said arguments from said first buffer to said second buffer; and,

(2) before a communication is returned from said kernel procedure,

j. upon completion of execution of said remote procedure, said remote procedure allocating a third buffer in said second memory space;

k. said remote procedure marshaling return arguments in said third buffer for returning from said remote procedure;

l. said remote procedure returning and passing a third reference to said third buffer to said kernel procedure;

m. said kernel procedure detecting said return of said remote procedure and allocating a fourth buffer in said first memory space;

n. said kernel procedure copying said return arguments contained in said third buffer into said fourth buffer;

o. said kernel procedure deallocating said third buffer;

p. said kernel procedure returning to said client procedure and passing a reference to said fourth buffer to said client procedure;

q. said client procedure detecting said returning to said client procedure and unmarshaling said arguments in said fourth buffer; and

r. said client procedure deallocating said fourth buffer and continuing execution,

wherein said first memory space is preallocated, prior to said calling of said remote procedure by said client procedure, to contain a buffer, a buffer pointer, and an allocation flag where said buffer pointer contains an address of said buffer

and said allocation flag contains a value representing one of an allocated state and a deallocated state for said buffer.

Previous Doc Next Doc Go to Doc#

Previous Doc Next Doc Go to Doc#
First Hit Fwd Refs

Generate Collection

L17: Entry 8 of 11 File: USPT Dec 2, 2003

DOCUMENT-IDENTIFIER: US 6658490 B1
** See image for Certificate of Correction **
TITLE: Method and system for multi-threaded processing

Abstract Text (1):
The present invention provides a method and system for multi-threaded processing that is an improvement over conventional systems. The system of the present invention comprises multiple threads of execution, multiple apartments, shared data, and a concurrency management component. The threads of execution run independently and each occupy one apartment. The apartments contain objects that have methods to perform operations. The shared data contains data that is accessible by all threads within the process. The concurrency management mechanism performs processing so that the multiple threads can execute concurrently in a reliable and robust manner. In an alternative system of the present invention, the threads are separate from the apartments and the threads execute within the apartments to perform processing. After performing the processing, the thread exits the apartment so that the apartment may be occupied by another thread.

Brief Summary Text (2):
The present invention relates generally to data processing systems and, more particularly, to multi-threaded processing within a data processing system.

Brief Summary Text (4):
A computer program that has been loaded into memory and prepared for execution is called a "process." A process comprises the code, data, and other resources such as files, that belong to the computer program. Each process in a data processing system has at least one thread which is known as the main thread. A thread comprises a pointer to a set of instructions, related central processing unit (CPU) register values, and a stack. A process can have more than one thread with each thread executing independently and keeping its own stack and register values. When a process has more than one thread, the process is said to be multi-threaded.

Brief Summary Text (5):
When a process performs multi-threaded processing, the threads can each perform a discrete unit of functionality. One advantage to multi-threaded processing is that threads can transfer information back and forth without having to cross process boundaries, which is expensive in terms of CPU processing time. Another advantage of multi-threaded processing is that when transferring data between threads of a single process, a reference to the data can be transferred instead of a copy of the data. Otherwise, if each discrete unit of functionality were implemented as a process, the data would typically have to be copied before being transferred to the destination process and this requires a significant amount of CPU processing time.

Brief Summary Text (6):
Two models used for performing multi-threaded processing by conventional systems are the free threading model and the lightweight process model. As shown in FIG. 1. the free threading model comprises a process 100 containing a number of objects 104, 108, 112, 116 and shared data 102. The term "object" refers to a combination of code ("Methods") and data. The code of the object typically acts upon the data

of the object. The shared data 102 is data that can be accessed by any object within the process 100. Each object in the free threading model has a lock 106, 110, 114, 118. The locks 106, 110, 114, 118 on the objects 104, 108, 112, 116 serialize access to each object so as to prevent contention problems. The shared data 102 has a semaphore 103 that serializes access to the shared data. In the free threading model. there are multiple threads and each thread can access any object 104, 108, 112, 116. Thus, the locks 106, 110, 114, 118 are necessary to prevent contention problems that may arise when more than one thread attempts to access an object.

Brief Summary Text (8):
FIG. 2 depicts a diagram of the lightweight process ("LWP") model for performing multi-threaded processing. In the LWP model, a process 200 contains a number of lightweight processes 202, 204, 206. Each lightweight process executes independently and contains procedures, data and variables. Therefore, a lightweight process is very similar to a thread. In the LWP model, each lightweight process can perform a discrete unit of functionality and can thus take advantage of the benefits of multi-threaded processing. However, the LWP model does not provide shared data and therefore when one lightweight process wants to communicate to another lightweight process, the data is typically copied before it can be sent. Performing a copy of the data before sending the data requires a significant amount of CPU processing time. In addition, since there is no shared data, data that all lightweight processes would like to access has to be maintained by a lightweight process. Such information includes location information for the procedures in the lightweight processes. Therefore, for example, whenever a lightweight process wants to determine the location of a procedure so that it may invoke the procedure, the lightweight process must communicate to the lightweight process that maintains the location information. This communication is costly in terms of CPU processing time.

Drawing Description Text (2):
FIG. 1 depicts the components utilized in the conventional free threading model for multi-threaded processing.

Drawing Description Text (3):
FIG. 2 depicts the components utilized in the conventional lightweight process model for multi-threaded processing.

Detailed Description Text (2):
The preferred embodiment of the present invention provides an improved method and system for performing multi-threaded processing. The preferred embodiment of the present invention is known as the "Apartment Model" for multi-threaded processing. The apartment model is an improvement over conventional systems by having multiple threads, shared data, and concurrency management on a per-thread basis. Each thread has one apartment in which it executes and each apartment contains objects. The shared data is data that is accessible by all threads within the process. The concurrency management of the preferred embodiment is implemented on a per-thread basis and manages the multiple threads so that each thread may concurrently execute in a reliable and robust manner.

Detailed Description Text (7):
A developer of an application program can conform to the apartment model for multi-threaded processing and thus take advantage of the benefits of the apartment model by utilizing apartments, shared memory, an RPC mechanism, and a CM mechanism. The developer utilizes apartments for their application program by dividing the application program into discrete units of functionality. Each discrete unit of functionality is then implemented as an apartment containing objects where each object performs functionality related to the discrete unit of functionality of the apartment. The developer utilizes shared memory by allocating a portion of memory within the address space of the application program to be accessible by all

apartments. The developer then defines which data within the application program should be placed into shared memory and develops semaphores to serialize access to the shared data. However, the semaphores may be provided to the developer as an operating system facility. The developer utilizes an RPC mechanism by either developing an RPC mechanism or by incorporating any of a number of well-known PPC mechanisms into the application program. The developer utilizes a CM mechanism in the application program by utilizing any acceptable CM mechanism such as the CM mechanism described in U.S. patent application Ser. No. 08/224,854, now abandoned, entitled "Concurrency Management in a Communication Mechanism." After utilizing the apartments, the shared memory, the RPC mechanism and the CM mechanism in the application program, the application program can then take advantage of the apartment model for multi-threaded processing.

Detailed Description Text (26):
The rental model is an alternative embodiment of the present invention where the threads are separate from the apartments in which the threads execute. Using the rental model, there are typically less threads than apartments. This has the advantage of reducing the overhead associated with having multiple threads. Another advantage to having threads separate from the apartments in which they execute is that concurrency management processing is reduced. That is, when a thread needs to execute within an apartment, if the apartment is unoccupied, there is no need to perform concurrency management when accessing the apartment. This saves the processing time associated with performing concurrency management. As shown in FIG. 14, a process 1400 utilizing to the rental model interacts with the operating system 310. The process 1400 has a number of threads 1402, 1404, 1406, including an RPC inbound thread 502, and a number of apartments 1408, 1410, 1412, and 1414. The apartments 1408, 1410, 1412, 1414 have objects 1416, 1418, 1420, 1422, 1424, 1426, 1428. The threads 1402, 1404, 1406 are independent paths of execution that receive messages from either another thread or the RPC inbound thread 502 via the operating system 310. The RPC inbound thread 502 receives messages (RPC messages) from another process via the operating system 310. Regardless of whether the thread is an RPC inbound thread or not, the processing performed by each thread is similar. Upon receiving a message, a thread will examine the message and determine the apartment in which the thread must execute in order to process the message. The thread then determines whether the apartment is occupied or unoccupied, and if the apartment is unoccupied, the thread occupies the apartment and processes the message. Otherwise, if the apartment is occupied, the thread sends a message to the thread occupying the apartment by posting a message to the operating system. This message will then be retrieved by the thread executing within the apartment and handled by the CM mechanism. That is, the message will be interleaved in the message queue and processed if the thread occupying the apartment is in a state capable of handling the message.

Previous Doc Next Doc Go to Doc#

WEST Search History

Hide Items | Restore | Clear | Cancel

DATE: Monday, February 07, 2005

Hide?	Set Name	Query	Hit Count
	DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=ADJ		
☐	L17	L15 and l3	11
☐	L16	L15 and l2	1
☐	L15	thread near8 examine near8 (request or process)	23
☐	L14	IN adj2 OUT adj2 COM adj4 buffer	0
☐	L13	20010330	30
☐	L12	L8 and l3	44
☐	L11	L8 and l3	44
☐	L10	L8 and l2	1
☐	L9	L8 and l4	1
☐	L8	(first adj buffer) same (second adj buffer)	7419
☐	L7	L6 and l4	1
☐	L6	request near8 buffer near8 retrieve	233
☐	L5	L4 and((buffer or buffering) near8 requst)	0
☐	L4	L3 and l2	34
☐	L3	multi adj (thread or threading or threaded)	5963
☐	L2	inter-thread adj2 communications	73
	DB=USPT; PLUR=YES; OP=ADJ		
☐	L1	5630074.pn.	1

END OF SEARCH HISTORY